A Presentation Manager Based on Application Semantics

Scott McKay
Symbolics, Inc.

William York
International Lisp Associates

Michael McMahon
Symbolics, Inc.

Abstract

We describe a system for associating the user interface entities of an application with their underlying semantic objects. The associations are classified by arranging the user interface entities in a type lattice in an object-oriented fashion. The interactive behavior of the application is described by defining application operations in terms of methods on the types in the type lattice. This scheme replaces the usual "active region" interaction model, and allows application interfaces to be specified directly in terms of the objects of the application itself. We discuss the benefits of this system and some of the difficulties we encountered.

Introduction

The function of a user interface is to display application information to the user in a clear, meaningful way and to provide a means of interacting with the application via that information. However, most existing UIMS's provide only a very weak coupling between the application's semantics and the visual entities with which the user interacts. For example, many systems provide the ability to cut and paste data between applications, but only by reducing the data to its lowest common denominator, typically text.

The lack in other systems of a good framework in which to connect application objects to their representation on the screen is hard on application programmers, because they must write application code at a relatively low level of abstraction, typically in terms of events (such as "mouse enters" and "button press") and callbacks (see [7], [10], and [11]). It is hard on the user, because it is often difficult for the application programmer to provide user interface consistency and good feedback. It is even hard on the programmer developing the UIMS itself. By using a model that is more closely coupled with the application, it is simple to provide a consistent user interface and such features as context-sensitive feedback and documentation, and the application programmer can write the user interface in terms of high-level objects rather than in terms of events.

By preserving the link between the user interface entities and their underlying application objects, the operations of an application can be performed directly on the semantically interesting components of the application (see [2], [12], and [18]). There would then be no need to "squeeze" the information through a textual representation that then needs to be reparsed to get back the original object. (The "live copy/paste" ideas in release 7.0 of the Macintosh (as described in [1]) address this issue as well.)

Motivation and Design

Wishing to address these issues, we set out to provide a richer framework for connecting user interface entities with the underlying semantically significant application objects they represent, while at the same time preserving the advantages of maintaining a separation between the code that implements the application's algorithms and the code that implements its user interface. We replaced the ubiquitous "active region" interaction model (see [7]) with a model that recognizes that each entity displayed on the screen is simply a visual representation of some application object, and that the user interface should be constructed in terms of operations on the application objects, not on the UIMS's data structures. In this model the application remains in control of the semantics of the user interface, but the details of the interactions, such as handling mouse motion and mouse clicks, are managed completely by the system.

Any application has its own set of user interface entities that arise naturally from the domain of the application. For example, an ECAD editor has device objects, such as wires, resistors, capacitors, and so on, each of which may have an associated user interface entity. The application must operate on the objects themselves, but the user interacts with the user interface entities. It is therefore necessary for the application to associate with each significant kind of application object a kind of user interface entity. Our system takes care of maintaining the association between application objects and the entities that

represent those objects.

Figure 1 shows the relationship between the internal and external representation of an application object and the input and output primitives (**accept** and **present**) which connect the representations of these objects.

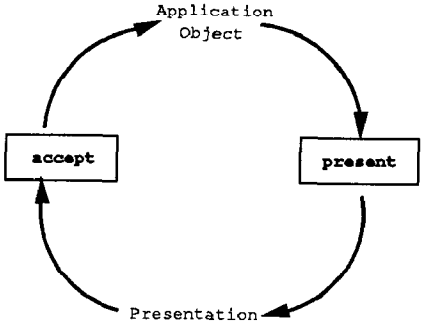

Figure 1. The relationship between an application object and its displayed representation.

The system we describe below, called Dynamic Windows, is part of Genera, the operating system used on Symbolics computers.

Semantic Types for Application Objects

One of the pieces missing from previous UIMS's has been the ability to specify directly the user interface behavior of, and relationships among, the application's user interface entities. We introduced a framework, in the form of a lattice of types, for specifying the behavior and relationships.

Each user interface object is given a *presentation type*, which is the semantic type of the object to be shown (that is, presented) to the user. A presentation type is defined by the application programmer for each object that participates in the user interface for an application. Presentation types form a lattice based on standard type-inheritance mechanisms such as those found in C++, SmallTalk, Lisp Flavors, and the Common Lisp Object System (CLOS) (see [3], [4], [14], [15], and [17]). This means that one presentation type may be defined as a subtype of one or more other types, sharing the basic characteristics of its supertype(s), while specializing other aspects of its behavior. For example, an application programmer writing an ECAD editor might define a general presentation type for devices, upon which they could build more specialized devices such as resistor and transistor.

The semantics of an application can be defined in terms of the relationships specified in the presentation type lattice. An operation that applies to a broad category of objects can be defined on a common ancestor supertype, while more specific operations can be defined directly on the appropriate subtree of the type lattice. For example, in an ECAD editor an operation such as "move device" might apply to all the device entities on the screen, but the "change capacitance" operation clearly applies only to capacitors.

We realized several kinds of advantages from using presentation types. First, because the presentation type system is based on the inheritance mechanisms found in ordinary object-oriented languages, application programmers have a familiar framework for declaring the relationships among the application objects. By placing each user interface entity in a type lattice, the application programmer can more easily reason about the behavior of the entities. Further, and more profoundly, user interface consistency becomes a by-product of the application's structure, rather than an artifact of graphic design. Such consistency provides a user interface that can be more easily understood by the end-user of an application (see [6]).

The presentation type lattice is richer than Common Lisp's basic data type lattice. One difference between presentation types and "normal" Lisp types is that presentation types have user interface-specific methods defined on them, such as methods that are responsible for parsing or printing an object of the type. Another difference is that the presentation type for an object may not be directly related to the primitive data type used by the application to implement the object data. For example, while an application may implement an ISO time object as a 32-bit integer, the ISO time presentation type really has nothing to do with 32-bit integers.

Output of Typed Objects

An application needs to display its objects to a user in such a way that it can track the association between the displayed representations of the objects and the objects themselves. Dynamic Windows maintains this association in data structures called *presentations*, which are the basic units of typed output.

A presentation is composed of three parts: the presentation type, the underlying application object, and the displayed representation of the object. This structure associates the application object with its displayed representation, and also associates the object with a presentation type.

Typed output revolves around two things: creating a presentation data structure and creating the display on the screen. Dynamic Windows provides a very general primitive for doing typed output, **with-output-as-presentation**, which provides a mechanism for associating an application object and its presentation type with an arbitrary piece of textual or graphical output. The application specifies the object and its presentation type, and then executes code that produces output. The system then creates a presentation that associates the output with the specified object and presentation type; this presentation is stored in the output history of the window on which the output is done.

For example, when an ECAD editor displayed a drawing of a circuit, it would create a presentation for each object in the circuit, which linked each object with its display on the screen. Figure 2 shows an example of the kind of application code (in Lisp) that would be used to draw a resistor as part of a circuit diagram. The code creates a presentation that contains the resistor object, the type of the object (resistor), and a pointer to the displayed representation of the resistor object, which is drawn by the application's drawing function, **draw-device**.

```
(with-output-as-presentation
  (:object R1
   :type 'resistor)
(draw-device R1 x-pos y-pos))
```

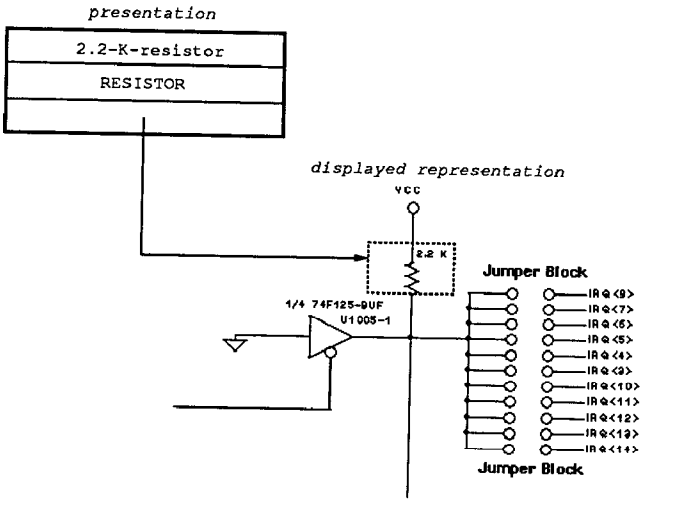

Figure 2. The presentation produced by some application code.

A more convenient primitive for doing typed output is **present**, which establishes a **with-output-as-presentation** context for the specified presentation type, and then invokes the method for displaying an object of that type. This allows the application programmer to display an object by simply specifying the object and its presentation type. For example, `(present #<RESISTOR ...> 'resistor)` creates a presentation whose object is the resistor object `#<RESISTOR ...>`, whose presentation type is **resistor**, and whose output on the screen is a drawing of the resistor. (In fact, **present** could be used in Figure 2 if the X and Y coordinates for a device object were attributes of the object itself.)

The display method for a presentation type, called its *printer*, registers a standard set of visual appearances for that type. The printer for a type takes an object of that type and displays some output that represents the object. It may define both graphical and textual representations for that type. Printers may be inherited from a supertype of the presentation type in the standard object-oriented way.

Dynamic Windows supports multiple views (as in [3]): a single presentation can have different appearances at different times. For example, a file name might be displayed as either a text string or as a file-folder shaped icon. Since presentations and the application objects that they represent are separate objects, the same application object can be displayed in multiple places at the same time.

Specification of Typed Objects as Input

Applications should interact with the user in terms of the application's objects. Since the system tracks the association between the application objects and their displayed representations on the screen, it remains only to provide a way for the user to specify an object as input to the application via the displayed representation of the object.

Whenever an application requests input of a certain presentation type from the user, the user can satisfy that request by selecting a presentation of that type with the mouse. The presentation type that the application is requesting is called the *input context*. The system automatically makes all presentations whose type matches the input context eligible as mouse-selectable input, and highlights eligible objects when the mouse is pointing at them. Furthermore, presentations whose type does not match the input context are not eligible. This context-based selectivity and highlighting provides feedback that assists the user in determining what user interface entities are selectable at any point in time (see [5]).

Any presentation whose type is a subtype of the input context is eligible as input. If an ECAD editor requests a device as input, any presentation whose type is based on the general device presentation type will be eligible, but if the application requests a transistor, only the transistors are eligible and not the whole set of devices. When reading input, the output view does not matter: only the presentation type is significant. For example, the visual representation of a file name might be a text string or a file-folder icon, but that visual representation has no bearing on whether the underlying object is or is not a file name.

The primitive for doing typed input is **accept**, which establishes the desired presentation type as the input context and then invokes the method for reading an object of that type. This allows the application programmer to request an object for input by simply specifying the desired presentation type. For example, `(accept 'resistor)` can be used to request a resistor device as input.

It is also useful for an application user to be able to specify objects by name from the keyboard. The method for a reading an object of a particular presentation type from the keyboard is called the *parser*. A type's parser specifies its input syntax, so that objects of that type can be read from the keyboard. The parser for a type may be inherited from a supertype of the type, and need not be defined at all.

Since **accept** acts as the dual of **present**, it is usually desirable for the parser (if there is one) to be the dual of the printer: the printer, when given an object, should produce a text string which, when parsed by the parser, produces the original object.

When **accept** invokes the parser for a presentation type, it arranges to handle mouse-driven input as well as keyboard-driven input. In the case where the user chooses to use the mouse to select the input, the parser is bypassed entirely.

The combination of both typed output and input contexts, and textual printers and parsers allows the creation of flexible mixed-mode user interfaces: users can fill in dialog boxes or construct command lines via an arbitrary, user-chosen combination of mouse clicks and keyboard input (see [8]).

Nested Presentations and Contexts

In addition to creating presentation types by defining completely new types or specializing existing ones, the programmer may build compound presentation types. For example, a Cartesian coordinate is a pair of real numbers; reading one from the keyboard might consist of reading a

floating-point number, discarding an intervening comma, and then reading another floating-point number. Printing a Cartesian coordinate object would consist of printing the two floating-point numbers separated by a comma. Note that this sort of aggregation is not an inheritance mechanism: Cartesian coordinates are neither subtypes nor supertypes of the floating-point type.

The most important compound type is the **command** presentation type, which is composed of a command name followed by the command's operands. This type is the building block for interacting with an application. We shall discuss this in more detail later.

Frequently, a presentation may contain other presentations inside itself, or a single user interface entity may represent more than one underlying application object. In order to support this, presentations may be nested. For example, in an ECAD editor, a device may be built up out of several other more basic devices connected by some wires. The device itself is a presentation, and the components contained in it are also presentations.

Input contexts may also be nested. In the case where presentations are nested, the input context determines which of the multiple presentations should be chosen when the user tries to select it with the mouse. For example, when an ECAD editor wishes to input a list of devices, the input context would be **(sequence device)**. The parser for the **sequence** type recursively establishes an input context for **device**. While in the inner context, presentations of both devices and sequences of devices will be eligible as input.

Gestures: Using the Mouse to Select Input

A *gesture* is a physical action performed by the user. In particular, a mouse gesture is an action performed using the mouse, such as clicking a button on the mouse. Gestures are used to provide a physical connection between an application operation and presentations having a particular presentation type.

Dynamic Windows provides a facility called logical gestures, which allows a physical gesture to be given a name. For example, by default, the logical gesture called **:select** is mapped to the physical gesture Mouse-Left (that is, "click the left-hand mouse button"). Application programmers should define operations on logical gestures instead of physical gestures, so that the mapping between logical gestures and physical mouse gestures can be tailored by the end-user if desired.

Dynamic Windows provides a standard set of logical gestures which, by convention, are used in particular ways by applications. The **:select** gesture is usually used to select an operand as is. The **:describe** gesture (which is mapped to Mouse-Middle by default) often has context-sensitive help associated with it. The **:menu** gesture (which is mapped to Mouse-Right by default) calls up a menu which contains the current set of applicable operations.

Type Coercion

Whenever an application is requesting input of a specific presentation type, presentations having that type (or any of its subtypes) may be selected with the mouse. However, it is often useful to be able to select a presentation that has another type which, although not strictly related by the type definitions, has some sort of a conceptual relationship or can be derived from the other. Dynamic Windows provides the ability to coerce an object having one presentation type to an object of another type. For example, if an ECAD editor is requesting a resistance value (in ohms), we might wish to satisfy that request by selecting a resistor with the mouse and extracting the resistance in ohms from the resistor.

It is not necessary to have translators that translate from a type to any of its subtypes, since this is handled automatically by the type inheritance mechanism.

All coercions are explicitly defined by the application programmer. The code that implements coercion is called a *translator*. A translator can be thought of as a method that specializes on a presentation type and a mouse gesture, and returns an object of another presentation type as its output. The application programmer specifies at translator definition time a *from* type, a *to* type, and a gesture. The following code implements a translator named **resistor-to-ohms** that extracts the resistance in ohms from a resistor object when the application is requesting a resistance as input and the user uses the **:select** gesture on the presentation of a resistor object:

```
(define-presentation-translator
  resistor-to-ohms
   (resistor resistance-in-ohms
    :gesture :select)
   (resistor)
  (resistor-resistance-in-ohms resistor))
```

Coercion is only available to mouse-driven input, because input from the keyboard does not have any particular presentation type associated with it. In effect, the presentation type of input from the keyboard is the same type as the input context.

Dynamic Windows provides a special kind of translator, called an *action*, that does not return a value but instead executes a side-effect. After the side-effect has been executed, the pending input request remains in effect. An example of an action that is context-sensitive (and whose behavior is mediated by the application) is one that might display a menu of possible completions while an application is requesting input of some presentation type. An example of actions that are context-independent (and are provided by the system as a standard service) are the textual cut and paste actions.

Selection of Translators

The set of applicable translators is determined by searching all of the known translators, matching the *from* type of each translator against the type of the presentation (what the mouse is pointing at), matching the *to* type of each translator against the input context (what the application is requesting), and matching each translator's gesture against the user's gesture.

Remember that Dynamic Windows does not simply do an exact type match: the presentation's type can be a subtype of the translator's *from* type, and the input context can be a supertype of the translator's *to* type. Nested presentations and nested input contexts make the matching algorithm more complex. If there are nested input contexts, the system searches outward from the innermost input context in order to find a context which has at least one applicable

handler. If there are nested presentations, the system chooses the innermost presentation which has at least one applicable translator.

The selection of translators can be refined by means of a *tester* that is called once a translator has passed the type discrimination tests. The tester serves as a filter, letting the application contribute more directly to the choice of translators.

Selecting a translator can also depend on the object returned by the translator. The input context can be "reduced" to a supertype combined with a predicate that tests the object returned by the translator. For example, consider a translator whose *to* type is **command**, and an input context of (**command :command-table "Global"**). The translator could return a command in any command table, but the input context only accepts commands in the command table named "Global". Since **command** is a supertype of (**command :command-table "Global"**) (not a subtype), the translator would not ordinarily be selected. However, since the input context "reduces" to **command** along with a predicate that tests whether the command is available in the "Global" command table, the translator tentatively matches and its body is executed. If the object returned by the translator satisfies the predicate, then the translator will be selected, otherwise it is ignored.

The algorithm for selecting a translator can result in the selection of many applicable translators. It is the responsibility of the application programmer to choose the gestures for the various translators in such a way that the number of "collisions" is kept to a minimum. However, such collisions can still occur, so Dynamic Windows provides a priority mechanism that allows the application programmer to decide which of several conflicting translators should be selected. An application user may also use the **:menu** gesture at any time to get a menu of all of the translators which apply at any given moment.

Because many translators might be applicable at any given time, Dynamic Windows provides additional feedback in the form of two "mouse documentation" lines at the bottom of the screen that are used to display what the current set of applicable translators are, what they do, and what physical mouse gestures they are assigned to.

Application Architecture

An application is composed of four major parts: the objects in the application and their current state, the code that implements the application itself, the layout of the application (screen real estate), and the operations that are invoked by the user via commands. This paper concentrates on the part of the application relevant to its user interface: the presentation types of the application's objects and the user-invokable operations on those objects.

In general, an application interacts with a user by reading a command and its operands from the user. It then executes the operation specified by the command and updates its internal data structures. Finally, the application updates the display on the screen. The code which implements this structure is called the *top-level command loop* of the application. For example, in an ECAD editor, the user may specify that he or she wishes to change the capacitance of a capacitor. The ECAD editor reads the command's operands, and then executes code that changes the state of the capacitor. After doing so, it updates the display of the circuit drawing to reflect the new capacitance.

To assist the application programmer in implementing this, Dynamic Windows provides a generic top-level command loop which is responsible for reading application commands, executing the code that implements the commands, and then calling an application-specific redisplay function to update the display.

Commands: the Interface to an Application's Operations

The interface to a particular application operation is called a *command*. The definition of a command specifies its operands and a small body of code that calls the application to perform the operation. Commands are stored in a per-application structure called a *command table*.

The programmer defines an application's commands by specifying for each command the presentation type of each operand and a small body of code which calls the application to perform the operation. The definition of a command can be thought of as a grammar which specifies how a command "sentence" is constructed from a "verb" (the actual command name, such as "Move Device" in a graphic editor), "nouns" (the objects on which the command operates, such as the devices in an ECAD editor), and modifiers. The body of the command, and the application code which implements an operation, need not know anything about how the operands of the command were read; the system guarantees that all the parameters passed to the application are objects of the specified type. This makes it simple to separate the interface of an operation from the algorithms that implement the operation.

Dynamic Windows provides a standard command reader (called the *command processor*) that reads an invocation of a command by sequentially calling **accept** for a command name and then for each operand. Since each operand is read with **accept**, the user can supply operands by using either the keyboard or the mouse. The command processor parses its input interactively, unlike traditional systems such as Unix, which parse the input only when the user types an end-of-line character. This interactive parsing is the basis for context-sensitive prompting and help facilities.

The command processor provides some default ways to read commands (for example, textual command lines, command menus, and dialogs) and is also responsible for executing commands once they have been read. An application user can choose which way he or she wishes to supply commands. Furthermore, an application programmer can change or extend the ways in which command sentences are read without having to modify the application itself. For example, Genera's Graphic Editor has a direct-manipulation user interface style for many of its common operations, such as shaping or moving a graphical entity. The direct-manipulation style was implemented in terms of ordinary translators.

Note that the concept of typed operands supports other styles of interactions besides the reading of commands. In particular, dialogs can also be specified in terms of the types of the objects to be read. The details of laying out

the dialog are managed by the system, unless the programmer specifies otherwise.

The **command** presentation type is a compound type: the command name and its operands are read via successive calls to **accept**. Since every invocation of a command is an instance of the **command** presentation type and the top-level loop of the application uses **accept** to read **command** objects, the application programmer can write translators that translate from some application-specific presentation type to an application command. Using such translators, the application programmer can build a user interface in which mouse gestures act as context-sensitive commands. For example, the following translator causes the "Change Resistance" command to be executed when the user clicks on a presentation whose type is **resistor**.

```
(define-presentation-translator
  change-resistance command
   (resistor
    :gesture :modify)
   (resistor)
  `(com-change-resistance ,resistor))
```

Building an Application

By breaking down the interface of an application into (typed) application objects and a set of operations on those objects, we are provided with a natural sequence for developing the user interface for an application. First the application programmer defines the presentation types that describe the significant objects of the application. Then the programmer defines the command interfaces to the operations on objects of those types. Finally, the programmer defines some translators that associate mouse gestures with commands on those objects. The commands and translators act as the links from a presentation to the application operations defined on the object that is represented by the presentation. The top-level command loop supplied by Dynamic Windows is responsible for both reading and executing the commands. The application itself is responsible only for implementing the operations for which the commands are the interface, and for displaying the user interface entities.

Design Details

In the above description of Dynamic Windows, we have omitted some details and simplified many things for purposes of clarity. In fact, what we implemented is richer and more complete than the description indicates.

Further Specialization of Presentation Types

The semantics of a presentation type can be refined via parameters to the type, called *data arguments*. The data arguments for a type affect what objects are instances of the presentation type. For example, the **integer** presentation type has as data arguments an upper and a lower bound. The syntax (that is, the visual appearance) of a presentation type can be refined via parameters to the type's methods, called *presentation arguments*. For example, the **integer** presentation has as presentation argument the radix in which it should be read and printed. Thus, a presentation type to specify the integers between zero and ten (inclusive) that should be read and printed in base 2 is `((integer 0 10) :base 2)`.

A simple form of type restriction is supported via the special **and** presentation type, which is used in conjunction with a **satisfies** clause. For example, the type `(and integer (satisfies oddp))` specifies the family of all odd integers.

A simple form of multiple inheritance is supported via the special **or** presentation type. This is particularly useful in specifying types that include a set of special tokens in addition to the basic type. For instance, the type `(or integer (member :all :none))` specifies a type that includes the two tokens "All" and "None" in addition to all of the integers.

Further Presentation Type Operations

The application programmer can supply definitions for a number of standard methods on presentation types in order to tailor the behavior of the type. We have already mentioned the parser and the printer. Other methods include the *describer*, which provides a description of the type for use in prompts and help messages, for example, "an integer between one and ten". The describer is used to support Dynamic Windows' context-sensitive help facility. Another method describes how items of this type are displayed in a dialog box. For example, the enumeration type (called **member**) might be displayed as Macintosh-style radio push-buttons, or the **boolean** type might be displayed as a check-box.

The typed output component of Dynamic Windows allows a set of *viewspecs* to be associated with a presentation. These viewspecs provide advice to the printer on how a presentation should be displayed, and can be changed at the behest of the application user. For example, the viewspecs for a file directory listing include a sorting predicate, such as whether the lines in the listing should be ordered alphabetically by file name or by the file's creation date. This sorting predicate can be changed by a menu. A standard translator provided by the system is one that expands the viewspecs of a presentation in order to show more or less detail.

Note that presentation arguments differ from viewspecs in that presentation arguments are specified by the application programmer in the application's code, but viewspecs can be changed by the application *user* in order to alter the appearance of a presentation.

Application-Building Aids

The typed input facility in Dynamic Windows provides a number of tools for writing parsers, such as a general-purpose completion facility. It also provides tools for tailoring context-sensitive help. For example, the definitions of commands and translators may specify code which provides documentation, prompting, and feedback in the mouse documentation lines.

Many prepackaged presentation types are provided by the system, such as integer, string, boolean, pathname, the **member** enumeration type, and so on. The **sequence** type allows the specification of a sequence of any other presentation types.

Genera itself provides tools for laying out entire applications.

Experiences and Evaluation

Implementation Details and Problems

Because we implemented Dynamic Windows in Common Lisp, we chose to use the Common Lisp type system as a model for the presentation type system. Ideally, the presentation type system would be simply an extension of the Common Lisp type system, but unfortunately we could not implement it this way for a number of reasons.

Since CLOS did not exist when we implemented Dynamic Windows, we used Flavors instead. Unfortunately, the Flavors system does not allow defining methods on primitive types and structures (such as integers and strings). Since presentation type methods such as parsers and printer require this, an extension to the Flavors type system had to be implemented. Since CLOS will allow class methods to be defined on primitive types and structures, this extension will no longer be necessary.

Neither Flavors nor CLOS provides any way to refine basic types, as is done by presentation type data arguments. Another extension to the type system was necessary to support this.

The handling of data and presentation arguments unfortunately cannot be handled at compile-time. For the sake of efficiency, the system maintains a cache of how data arguments might affect presentation type inheritance. Furthermore, the selection of application translators for a given presentation type in a given input context can be very expensive, so the system maintains a cache for this as well. However, there is no advertised convention for when to clear these caches, which can result in mysterious problems.

In Dynamic Windows, translators are defined globally. When there are many translators, the selection of applicable translators can be time-consuming. Storing an application's translators in its command table could greatly reduce the amount of time required to choose applicable translators.

Since we designed Dynamic Windows to assist in the development of programs written in Lisp, it provides a feature that allows the Lisp data-type of the object in a presentation to be treated as though it is the presentation type. This allows many translators to be useful on presentations of "raw" Lisp objects, but at the cost of greatly increasing the number of translators that might be considered in any input context. This increase causes noticeable performance problems.

Right now, the parsers and printers for compound presentation types are specified procedurally rather than declaratively. Furthermore, since the parser and the printer are duals, the procedures that implement the parser and printer for a compound type usually parallel each other very closely. In practice, this has proved to be annoying and error-prone. A better scheme would be to specify the syntax of compound presentation types with some sort of simple grammar.

The support for multiple views could be made much more powerful by allowing a separate specification for a graphical view (if there is one) and a textual view (if there is one). The system could then decide which view to use based on the type of output device or the context of the application. Right now, there is no way for application users to specify a general preference for the kind of view they wish to see.

The users of Dynamic Windows have cited the lack of support for graphical input facilities to match the support for graphical output. For example, the system could provide a more complete set of "gadgets" such as dials, gauges, and sliders. In fact, gadgets could be implemented as a form of multiple views.

Observations

There are two complementary observations one might make regarding Dynamic Windows as it presently stands.

- *"The Unix observation"*: In an application that uses a command line interface style where all objects have textual representations, there is little need for the ability to click on a presentation to recover its exact semantics. With a carefully designed set of cut and paste commands, one can just as well use the mouse to click on text and have a parser recover the semantics. In fact, Dynamic Windows does support textual cutting and pasting, but it is much more powerful to share high-level data structures among applications. Having to resort to cutting and pasting also means that application writers must write parsers and printers that might otherwise be unnecessary. Also, it may not even be possible to recover the exact semantics of an object from a textual representation.
- *"The Macintosh observation"*: In Genera, many applications associate presentation semantics only with pieces of text (as opposed to graphics or icons). This gives those applications an archaic appearance when compared to systems such as the Macintosh. This bias is not inherent to Dynamic Windows, but is an artifact of the age of the applications, many of which were written before this system was in place. In fact, Dynamic Windows is ideally suited for graphical user interfaces because it is possible for the user to request an operation on an exact application object without having to resort to specifying the object by some textual representation.

Comparison with Conventional Toolkits

Presentation types are not like conventional user interface widget toolkits (see [7] and [16]). Conventional widgets are intended as layout tools, and to insulate the applications programmer from having to deal directly with the underlying window system. Such widgets operate at a fairly low level: the application yields control to the widget manager, which waits for events (such as "mouse enters", "mouse leaves", "button press", and so on) and communicates with the applications in terms of those events by using callbacks or resources. All the semantics of the application are implemented via the callbacks; the widgets themselves are usually careful not to supply any sort of semantics.

In Dynamic Windows, the top-level command loop of the application remains in control and requests events from the system. These events take the form of actual application objects and commands. Events such as "mouse enters" and "button press" are hidden from the application, unless the application explicitly requests that level of detail.

Presentation types do not act as a framework for laying out the overall appearance of the user interface (such as in [9] or [13]). There are higher-level tools in Dynamic

Windows to indicate that the application programmer wishes to display, for example, a menu or a dialog, but the actual layout of the menu or dialog is typically managed automatically. Genera provides other tools for laying out the overall appearance of an entire application.

Conclusions

When we designed Dynamic Windows, we set out to create a system which would allow programmers to develop good user interfaces for applications, in which the interactions with the user interface could be describe in terms of the objects in the applications using the object-oriented paradigms with which we were familiar. We wanted to free the application programmer from the burden of writing the user interface in very low-level terms, so that the programmer would be free to concentrate of the application itself or would be able to experiment with alternate user interface styles without having to rewrite any of the application. We wanted the resulting applications to have consistent, predictable, robust user interfaces. In these goals, we succeeded. Unfortunately, the performance goals we set for ourselves have never been completely met: the performance of Dynamic Windows is good, but not as good as the performance provided by more conventional user interface toolkits. We also wanted to provide programmers a good set of user interface design tools, but these tools have yet to come to full fruition.

In practice, Dynamic Windows has proved to be a powerful and flexible tool for easily constructing applications and their interfaces. For example, it was used to construct the interface to Genera's hypertext documentation system, Concordia [19]. The third author built Genera's font editor and graphical editor (which uses a hybrid direct-manipulation user interface style) using it. A graphically oriented interface for Genera's debugger is also built on Dynamic Windows. The first author built a simple direct-manipulation business graphics package (including spreadsheets, charts, and graphs) using this system.

The original version of Dynamic Windows was developed in Symbolics Common Lisp during 1986 and 1987, an initial version was delivered to Symbolics customers in 1987, and an improved version was released in early 1988. The development of a portable Common Lisp successor to Dynamic Windows based on CLOS, called CLIM (Common Lisp Interface Manager), is now in progress. The design of CLIM builds upon the experiences gained with Dynamic Windows, and addresses the problems we have discovered in Dynamic Windows. Applications written using CLIM will be portable across hosts running various window systems, such as X, QuickDraw, and MicroSoft Windows.

Acknowledgments

The authors would like to thank Dave Moon, Dennis Doughty and John Aspinall for excellent review comments. Jan Walker, Polle Zellweger, and Jock Mackinlay provided crucial guidance on how to structure the paper to make it readable.

The design and implementation of Dynamic Windows was a large joint effort that could not have been done without the contributions of many engineers at Symbolics; many thanks to all of them.

References

1. *System Software Release 7.0 - Data Publication Manager.* System Software Release 7.0 (preliminary) edition, Apple Computer, Inc., 1989.

2. E. C. Ciccarelli. Presentation Based User Interfaces. Tech. Rept. AI-TR 794, MIT A.I. Laboratory, 1984.

3. A. Goldberg and D. Robson. *SmallTalk-80: the Language and its Implementation.* Addison-Wesley, 1983.

4. S. Keene. *Object Oriented Programming in Common Lisp.* Addison-Wesley, 1989.

5. H. Lieberman. "There's more to Menu Systems than Meets the Eye". *Computer Graphics: SIGGRAPH 1985 Conference Proceedings* , (July 1985), pp. 181-189.

6. H. Lieberman. Using Prototypical Objects to Implement Shared Behavior in Object-Oriented Systems. Proc. OOPSLA 1986, ACM, 1986.

7. J. McCormack, *et al.*. *X Toolkit Intrinsics - C Language Interface.* MIT, 1988.

8. M. McMahon. A Practical System for Managing Mixed-mode User Interfaces. Unpublished. Forthcoming.

9. B. A. Myers. "Creating Interaction Techniques by Demonstration". *IEEE Computer Graphics and Applications* , (September 1987), pp. 51-60.

10. A. Nye. *Xlib Programming Manual.* O'Reilly & Associates, 1988.

11. A. Nye (editor). *Xlib Reference Manual.* O'Reilly & Associates, 1988.

12. D. Olsen. "ACM SIGGraph Workshop on Software Tools for User-Interface Management". *Computer Graphics* , (April 1987), pp. 71-147.

13. K. J. Schmucker. "MacApp: An Application Framework". *Byte* , (August 1986), pp. 189-193.

14. G. L. Steele, Jr.. *Common Lisp: the Language.* Digital Press, 1984.

15. B. Stroustrup. *The C++ Programming Language.* Addison-Wesley, 1986.

16. R. Swick, *et al.*. *X Toolkit Athena Widgets - C Language Interface.* MIT, 1988.

17. *Symbolics Common Lisp - Language Concepts.* Genera 7.0 edition, Symbolics, Inc., 1986.

18. P. Szekely. "Modular Implementation of Presentations". *Proc. SIGCHI+GI 1987* , (April 1987), pp. 235-240.

19. J. Walker. "Supporting Documentation Development with Concordia". *IEEE Computer* , (January 1988), pp. 48-59.